

Lab4- Stream processing with Spark

Outline

This lab will teach you the basic of stream processing with Spark. As soon as an application compute something with business value (for instance customer activity), and new input arrive continuously, companies will want to computer this result continuously too. Spark make it's possible to process stream with the Structured Streaming Api. This lab will teach you the basics of this Spark Api. Because the Structured Streaming Api and the DataFrame Api shared the same syntaxes, so all the syntaxes in lab 1 car be used here, this lab will be in two parts :

- The first part will be a presentation of the basics of Spark's Structured Streaming Api with simple data
- The second part will present how to use tweets to sentiment analysis in real time.

Spark cluster creation in AWS

First: **DO NOT FORGET TO TURN YOUR CLUSTER OFF A THE END OF THIS TUTORIAL!**

Instructions are at the beginning of lab1

DO NOT FORGET TO TURN YOUR CLUSTER OFF A THE END OF THIS TUTORIAL!

Configuration of the notebook

```
1 # Configuraion
2 # The user pay the data transfer
3 spark._jsc.hadoopConfiguration().set("fs.s3.useRequesterPaysHeader","true")
4
5 # Set the number of shufflre partitions
6 spark.conf.set("spark.sql.shuffle.partitions", 5)
7
8 # Import all the needed library
9 from time import sleep
10 from pyspark.sql.functions import from_json, window, col, expr
11 from pyspark.sql.types import StructType, StructField, StringType,
12 IntegerType, ArrayType, TimestampType, BooleanType, LongType, DoubleType
```

Explications:

- `spark._jsc.hadoopConfiguration().set("fs.s3.useRequesterPaysHeader","true")` : like in lab2, you will be charged for the data transfer. without this configuration you can't access the data.
- `spark.conf.set("spark.sql.shuffle.partitions", 5)` : set the number of partitions for the shuffle phase. A partition is in Spark the name of a bloc of data. By default Spark use 200 partitions to shuffle data. But in this lab, our mini-batch will be small, and to many partitions will lead to performance issues.

```
1 spark.conf.set("spark.sql.shuffle.partitions", 5)
```

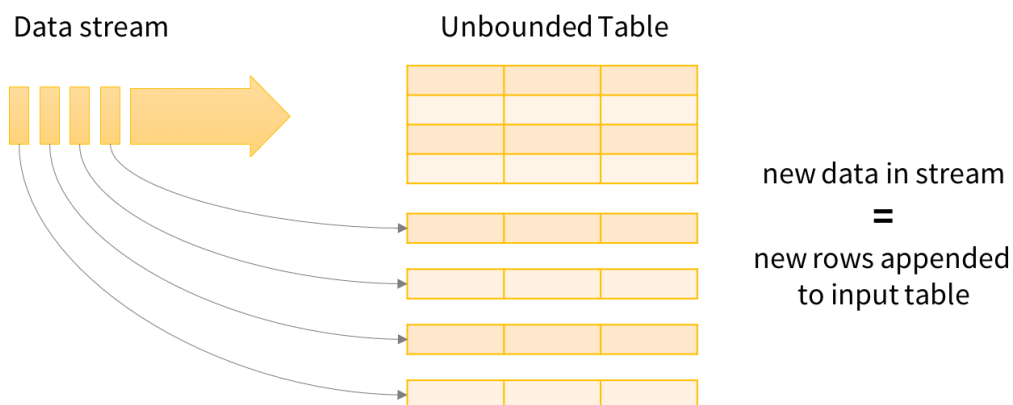
☹️ La phase de *shuffle* consiste à ré-ordonner les données selon leur clef, entre une étape *map* et une étape *reduce*. Par exemple, si vous comptez le nombre d'observations dans le groupe dans chaque groupe *g*, l'étape *map* consiste à compter le nombre de membres du groupe dans un bloc: {g1:5, g2:10, g4:1, g5:3} pour un bloc, {g1:1, g2:2, g3:23, g5:12} pour un autre. L'étape *shuffle* consiste à réorganiser les résultats intermédiaires en de nouveaux blocs, avec les mêmes clés au même endroit: {g1:5, g1:1, g2:10, g2:2} pour un bloc, {g4:1, g5:3, g3:23, g5:12}. pour un autre. De cette façon, l'étape *reduce* est considérablement accélérée.

- 📄 Import all needed library

```
1 from time import sleep
2 from pyspark.sql.functions import from_json, window, col, expr
3 from pyspark.sql.types import StructType, StructField, StringType,
  IntegerType, ArrayType, TimestampType, BooleanType, LongType,
  DoubleType
```

1. ✨ Spark and stream processing 📁

Stream processing was gradually incorporated in Spark. In 2012 Spark Streaming and the DStreams API was added to Spark. This made it possible to stream processing to use high-level operator like *map* and *reduce*. Because of its implementation this API has some limitations. Thus, in 2016 a new API was added, the Structured Streaming API. This API is directly build built on DataFrame, unlike DStreams. This has an advantage, you can process your stream like static data. Of course there is some limitation, but the core syntaxes is the same. You will chain transformations, because each transformation take a DataFrame as input and produce a DataFrame as output. The big change is there is no action at the end, but a [output sink](#).



Data stream as an unbounded Input Table

Stream processing is the act to process data in real-time. When a new observation is available, it is processed. There is no real beginning nor end to the process, and there is no "result". The result is update in real time, hence multiple version of the results exist. For instance, you count how many tweet about cat are posted in twitter every hour. Until the end of an hour you do not have your final result. But even at this moment, your result can change. Maybe some

technical problem created some latency and you will get some tweets later. If this case, maybe it's not a big deal, but in some cases it can be.

Some common use cases of stream processing are :

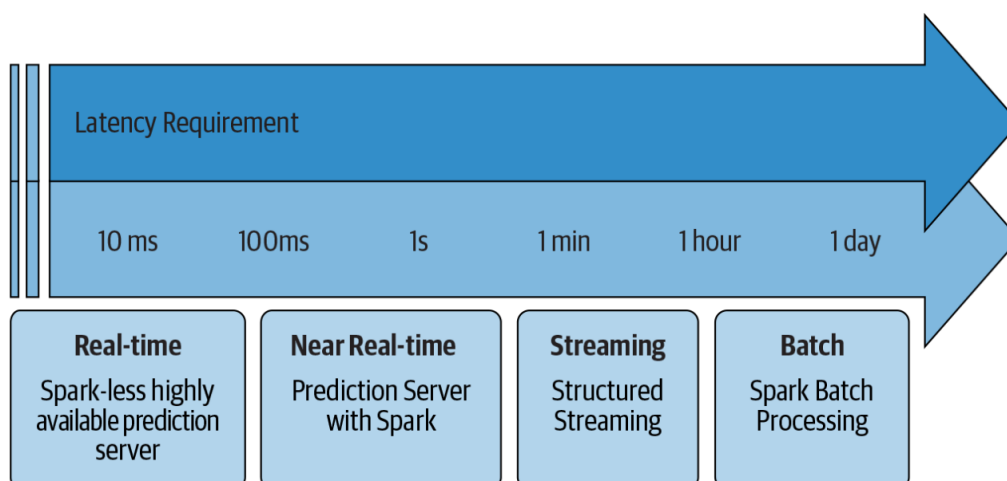
- **Notifications and alerting** : real-time bank fraud detection ; electric grid monitoring with smart meters ; medical monitoring with smart meters, etc.
- **Real time reporting**: traffic in a website updated every minute; impact of a publicity campaign ; stock option portfolio, etc.
- **Incremental ELT (extract transform load)**: new unstructured data are always available and they need to be processed (cleaned, filter, put in a structured format) before their integration in the company IT system.
- **Online machine learning** : new data are always available and used by a ML algorithm to improve its performance dynamically.

Unfortunately, Stream processing has some issues. First because there is no end to the process, you cannot keep all the data in memory. Because your memory is limited. Second, process a chain of event can be complex. How do you raise an alert when you receive the value 5, 6 and 3 consecutively ? Don't forget you are in a distributed environment, and there is latency. Hence, the received order can be different from the emitted order.

Spark offer two ways to process stream, one **record at a time**, or processing micro batching (processing a small amount of line at once).

- **one record at a time** every time a new record is available it's processed. This has a big advantage, the **latency is very low**. But there is a drawback, the system can not handle too much data at the same time (low throughput)
- as for **micro batching** it process new records every τ seconds. Hence record are not process really in "real-time", **the latency will be higher, and so the throughput**. Unless you really need low latency, make it your first choice option.

🕒 To get the best decision between latency / throughput, a good practice is to decrease the micro-batch size until the mini-batch throughput is the same as the input throughput. Then increase the size to have some margin



To understand why processing one record at a time has lower latency than throughput, imagine a restaurant. Every time a client orders something the chef cooks its order independently of the other current orders. So if two clients order pizza, the chef makes two small doughs, and cook them individually. If clients come slowly, the chef can finish each order before a new client comes. The latency is the lowest possible the chef is idle when a client comes. Now imagine restaurant where the chef processes the orders by batch. Each time he waits some minutes to gather all the orders then he mutualize the

cooking. If there is 5 pizza orders, he only do one big doughs, divide it in five, add the topping then cook all five at once. The latency is higher because the chef wait, but so the throughput because he can cook multiple thing at once.

2. The basics of Spark's Structured Streaming

2.1. The different sources for stream processing in Spark

In lab 2 you discovered Spark DataFrame, in this lab you will learn about [Structured Streaming](#), the Spark object to handle stream data. It's a stream processing framework built on the Spark SQL engine, and it use the existing structured APIs in Spark. So once you define a way to read a stream, you will get a DataFrame. Like in lab2 ! **So all transformations presented in lab2 are still relevant in this lab.**

Spark Streaming supports several input source for reading in a streaming fashion :

- [Apache Kafka](#) an open-source distributed event streaming platform (not show in this lab)
- Files on distributed file system like HDFS or S3 (Spark will continuously read new files in a directory)
- A network socket : an end-point in a communication across a network (sort of very simple webservice). It's not recommend for *production* application, because a socket connection doesn't provide any mechanism to check the consistency of data.

Defining an input source is like loading a DataFrame but, you have to replace `spark.read` by `spark.readStream`. For instance, if I want to open a stream to a folder located in S3 you have to do

```
1 my_first_stream = spark\  
2   .readStream\  
3   .schema(schema_tweet)\  
4   .json("s3://my-awesome-bucket/my-awesome-folder")
```

The major difference with lab2, it is Spark cannot infer the schema of the stream. You have to pass it to Spark. There is two ways :

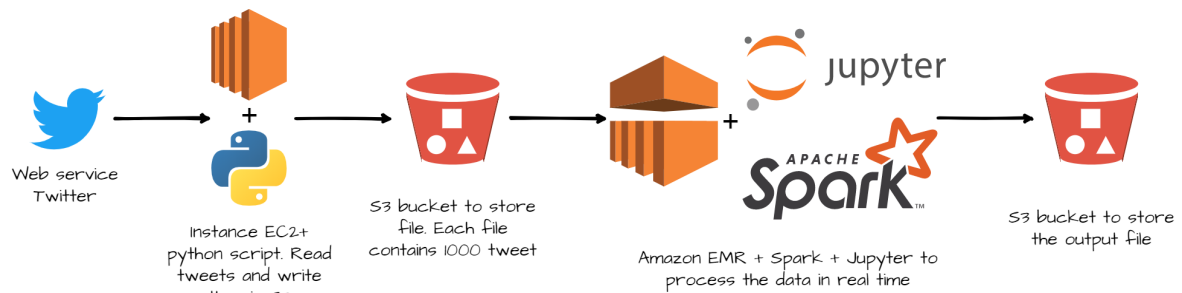
- A reliable way : you define the schema by yourself and gave it to Spark
- A quick way : you load one file of the folder in a DataFrame,, extract the schema and use it. It works, but the schema can be incomplete. It's a better solution to create the schema by hand and use it.

For Apache Kafka it's a slightly more complex, (not used today, it's just for your personal knowledge):

```
1 my_first_stream = spark\  
2   .readStream\  
3   .format("kafka")  
4   .option("kafka.bootstrap.servers", "host1:port1, host2:port2 etc")  
5   .option("subscribePattern", "topic name")  
6   .load()
```

Why is a folder a relevant source in stream processing ?

Previously, in lab 2, you load all the files in a folder stored in S3 in Spark. And it worked pretty well. But this folder was static. Its content doesn't change. But in some cases, new data are constantly written into the folder. For instance, here is the complete workflow of the second part of the lab :



A script python is running in a EC2 machine and constantly add file to a S3 buckets. Every 20 seconds, a new file is added to the bucket. If you use DataFrame like in lab 2, your process cannot proceed new files. You should relaunch your process. But with Structured Streaming Spark will dynamically load new files.

Because cloud providers offer high durability, cheap storage solutions with tools to interact easily with there storage solution, in many cases, it easier to have scripts which write in a bucket and others which read from it that create some complex architectures to stream data.

The remaining question is, why don't we connect Spark to the twitter webservice directly ? And the answer is : we can't. Spark cannot be connected to a webservice directly. You need some code between Spark and your webservice. There are multiple solutions, but an easy and reliable solution is to write tweet to s3 (because we use AWS services, if you use Microsoft Azure, Google Cloud Platform or OVH cloud replace S3 by their storage service).

Hand-on 1 : open a stream

In the first part of this lab, you will use IoT (*Internet of Things*) data. The dataset came from the [Heterogeneity Activity Recognition Data Set](#) and consists of smartphone and smartwatch sensors readings from variety of devices. For example, here is some readings

```
1 {"Arrival_Time":1424686735175
2  ,"Creation_Time":1424686733176178965
3  ,"Device":"nexus4_1"
4  ,"Index":35
5  ,"Model":"nexus4"
6  ,"User":"g"
7  ,"gt":"stand"
8  ,"x":0.0014038086
9  ,"y":5.0354E-4
10 ,"z":-0.0124053955}
```

- Define a variable with this schema

```

1 StructType()\
2     .add('Arrival_Time',    TimestampType(),    True)\
3     .add('Creation_Time',    TimestampType(),    True)\
4     .add('Device',          StringType(),    True)\
5     .add('Index',           LongType(),    True)\
6     .add('Model',           StringType(),    True)\
7     .add('User',            StringType(),    True)\
8     .add('_corrupt_reccord',StringType(),    True)\
9     .add('gt',              StringType(),    True)\
10    .add('x',                DoubleType(),    True)\
11    .add('y',                DoubleType(),    True)\
12    .add('z',                DoubleType(),    True)

```

- Crte a stream to this s3 bucket : `s3://spark-lab-input-data-ensai20202021/Iot/`. Name it `iot_stream`

🤔 Nothing happen ? It's normal ! Do not forget, Spark use lazy evaluation. It doesn't use data if you don't define an action. For know Spark only know how to get the stream, that's all.

- In a cell just execute `iot_stream`. It should print the type of `iot_stream` and the associated schema. You can see you created a DataFrame like in lab2 !
- To print the size of your DataFrame with this piece of code :

```

1 iot_query = iot_stream\
2     .writeStream\
3     .queryName("iot_stream")\
4     .format("memory")\
5     .start()
6
7 for _ in range(10): # we use an _ because the variable isn't use.
8     sleep(3)        You can use i if you prefere
9     spark.sql("""
10         SELECT count(1) FROM iot_stream""").show()
11 iot_query.stop() #needed to close the stream
12

```

2.2 How to output a stream ?

Remember, Spark has two types of methods to process DataFrame:

- Transformations which take a DataFrame has input and produce an other Dataframe
- And actions, which effectively run computation and produce something, like a file, or a output in you notebook/console.

Stream processing looks the same as DataFrame processing. Hence, you still have transformations, the exact same one that can be apply on classic DataFrame (with some restriction, for example you can not sample a stream with the `sample()` transformation). The action part is a little different. Because a stream runs continuously, you cannot just print the data at the end of the process, because there is no end by definition, so your output need to be update constantly (or at least periodically). To tackle this issue, Spark proposes different [outputs sinks](#). An output sink is a possible output for your stream. The different output sink are (this part came from the official Spark [documentation](#)) :

- **File sink** - Stores the output to a file. The file can be stored locally (on the cluster), remotely (on S3). The file format can be json, csv etc

```
1 writeStream
2   .format("parquet")           // can be "json", "csv", etc.
3   .option("path", "path/to/destination/dir")
4   .start()
```

- **Kafka sink** - Stores the output to one or more topics in Kafka.
- **Foreach sink** - Runs arbitrary computation on the records in the output. It does not produce an DataFrame. Each processed lines lost

```
1 writeStream
2   .foreach(...)
3   .start()
```

- **Console sink (for debugging)** - Prints the output to the console standard output (stdout) every time there is a trigger. This should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory after every trigger. *Sadly console sink does not work with jupyter notebook.*

```
1 writeStream
2   .format("console")
3   .start()
```

- **Memory sink (for debugging)** - The output is stored in memory as an in-memory table. This should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory. Hence, use it with caution. Because we are in a simple lab, you will use this solution. But keep in mind it's a very bad idea because data must fit in the the ram of the driver node. And in a big data context it's impossible. Because it's not a big data problem if one computer can tackle it.

```
1 writeStream
2   .format("memory")
3   .queryName("tableName") # to request the table with spark.sql()
4   .start()
```

We just talked where we can output a stream, but there is another question, how ?

To understand why it's a issue, let's talk about two things that spark can do with stream : filter data and group by + aggregation

- **Filter** : you process is really simple. Every time you get a new data you just compute a score and drop the row if the score is less than a threshold. Then you write into a file every kept row. In a nutshell, you just append new data to a file. Spark does not have to read an already written row.
- **Group by + aggregation** : in this case you want to group by your data by key than compute a simple count. Then you want to write the result in a file. But now there is an issue, Spark need to update some existing rows in your file every time. But is your file is stored in HDFS, it's impossible to update in a none append way a file. In a nutshell, it's impossible to output in a file your operation.

To deal with how output stream, Spark proposes 3 mode. And you cannot use every mode with every output sink, with every transformation. The 3 modes are :

- **Append mode (default)** - This is the default mode, where only the new rows added to the Result Table since the last trigger will be outputted to the sink. This is supported for only those queries where rows added to the Result Table is never going to change. Hence, this mode guarantees that each row will be output only once (assuming fault-tolerant sink). For example, queries with only `select`, `where`, `map`, `flatMap`, `filter`, `join`, etc. will support Append mode.
- **Complete mode** - The whole Result Table will be outputted to the sink after every trigger. This is supported for aggregation queries.
- **Update mode** - (Available since Spark 2.1.1) Only the rows in the Result Table that were updated since the last trigger will be outputted to the sink. More information to be added in future releases.

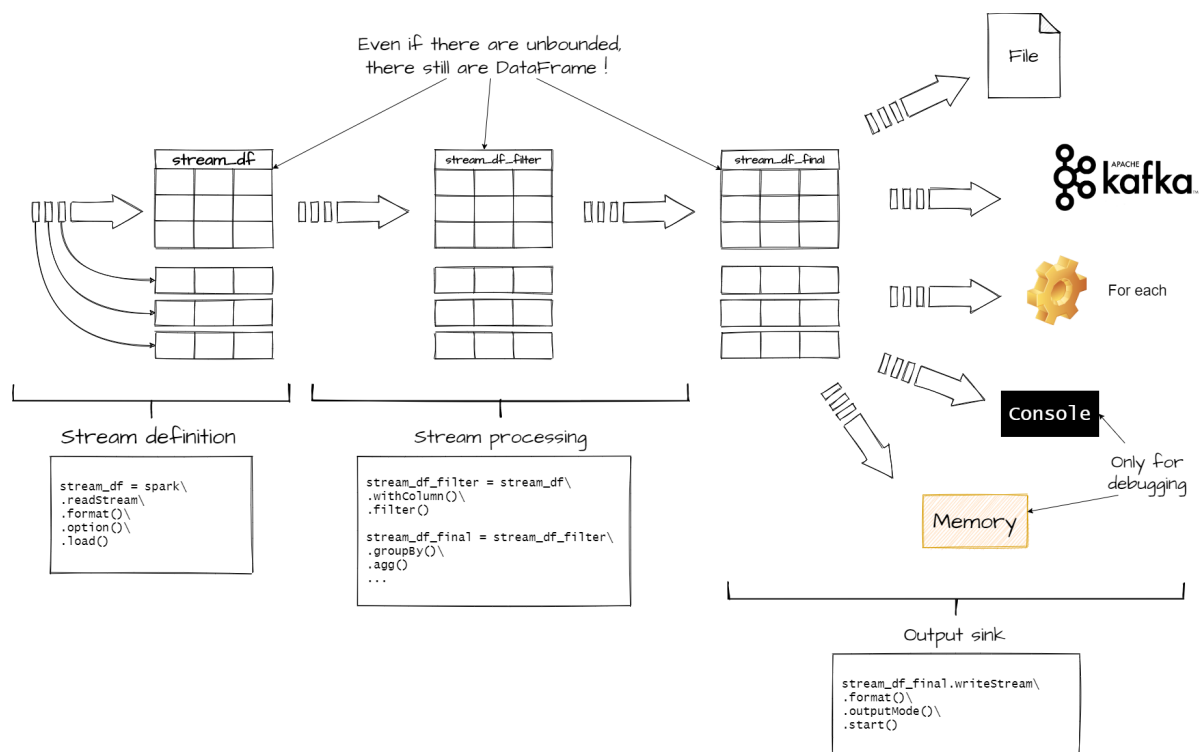
Sink	Supported Output Modes
File Sink	Append
Kafka Sink	Append, Update, Complete
Foreach Sink	Append, Update, Complete
ForeachBatch Sink	Append, Update, Complete
Console Sink	Append, Update, Complete
Memory Sink	Append, Complete

To sum up : To output a stream you need

- DataFrame (because once load a stream is a DataFrame)
- A format for your output, like console to print in console, memory to keep the Result Table in memory, json to write it to a file etc
- A mode to specify how the Result Table will be updated.

For instance for the memory sink

```
1 memory_sink = df\  
2 .writeStream\  
3 .queryName("my_awesome_name")\  
4 .format('memory')\  
5 .outputMode("complete" or "append")\  
6 .start() #needed to start the stream
```

Hand-on 2 : output a stream

Activity count

- Compute a DataFrame that will group an count data by the `gt` column. Name your DataFrame `activity_count`
- Use this DataFrame to create a output stream with the following configuration :
 - Memory sink
 - Complete mode (because we are doing an agregation)
 - Name you query `activity_count`
- Then past this code

```
1 for _ in range(10): # we use an _ because the variable isn't use. You
2   sleep(3)          can use i if you prefere
3   spark.sql("""
4     SELECT * FROM activity_count""").show()
5   activity_query.stop() #needed to close the stream
```

After 30 seconds, 10 table will appeared in your notebook. Each table are the contain of `activity_count` at a certain time. The `.stop()` method close the stream.

In the rest of this tutorial, to will need two steps to print data :

1. Define a stream with a memory sink
2. Request this stream with the `spark.sql()` function

Instead of a for loop, you can just write you `spark.sql()` statement in a cell and rerun it. In this case you will need a third cell with a `stop()` method to close your stream.

For instance:

- Cell 1

```

1 my_query = my_df\
2   .writeStream\
3   .format("memory")\
4   .queryName("query_table")\
5   .start()

```

- Cell 2

```

1 spark.sql("SELECT * FROM query_table").show()

```

- Cell 3

```

1 iot_data_memory.stop()


```

✗ Count row with null value

- Add a column `error` to your DataFrame. This column equal True if `device`, `index`, `model`, `user` and `gt` are all null. Else it's false. Use the `withColumn` transformation to add a column
- Group and count by the `error` column
 - Print some results

Stream processing basics

Hand-on 3 : transformations on stream

-  Filter stream all row with a null value then group and count data by `gt`.
 - For this filter, you will use the `na.drop("any")` transformation. The `na.drop("any")` drop every line with a null value in at least one column. It's simpler than using a `filter()` transformation because you don't have to specify all the column. For more precise filter you can use `na.drop("any" or "all", subset=list of col)` (`all` will drop rows with only null value in all columns or in the specified list).
- ▼ Column creation and filtering :
 - Define a new column, name `is_stair_activity`. This column is equal to `True` if the `gt` contains "stairs", else it's equal to `False`. To do so use the `withColumn()` transformation, and the `expr()` function. It take as input a SQL expression. You do not need a full SQL statement (`SELECT ... FROM ... WHERE ...`) but just an SQL expression that return True or False is `gt` contains "stairs" ([for some help](#))
 - Only keep `gt`, `model`, `arrival_time`, `creation_time`

Hand-on 4 : Aggregation and grouping on stream

- Count the number of different users.
- Group by `user` and compute the average, min and max of `x`, `y` and `z` .
 - Use the `groupBy()` and `agg()` transformations
- Compute the average of `x`, `y` and `z` :
 - across all `gt` and `user`

- for each `user` across all `gt`
- for each `gt` across all `user`
- for each `gt` and each `user`

To do so, replace the `groupBy()` transformation by the `cube()` one. `cube()` group compute all possible cross between dimensions passed as parameter. You will get something like this

gt	model	avg(x)	avg(y)	avg(z)
sit	null	some_value	some_value	some_value
stand	null	some_value	some_value	some_value
...
walk	nexus1	some_value	some_value	some_value
null	nexus1	some_value	some_value	some_value
null	null	some_value	some_value	some_value

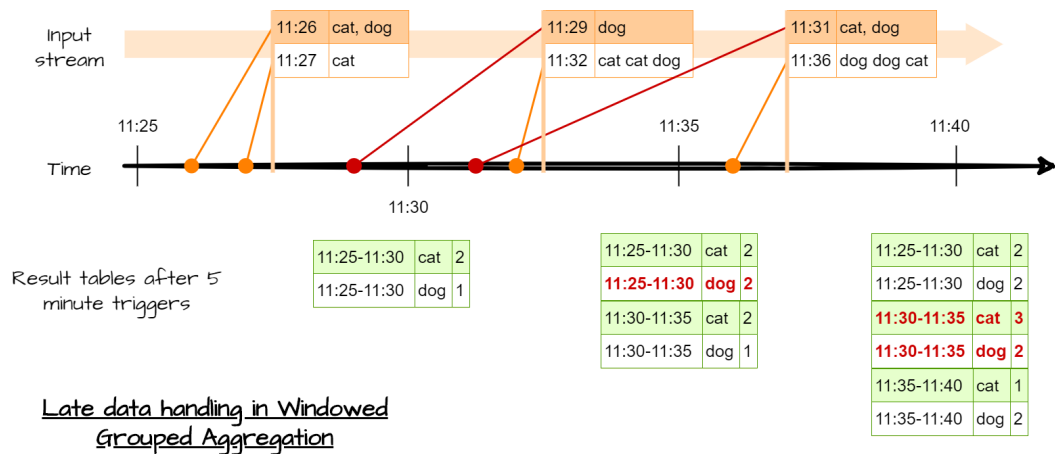
A `null` value mean this dimension wasn't use. For instance, the first row give the average when `gt==sit` independently of the `model`. The before last row give averages when `model==nexus1` independently of the `gt`. And the last row give the averages for the full DataFrame.

Event-time processing

Event-time processing consists in processing information with **respect to the time that it was created, not received**. It's a hot topic because sometime you will receive data in an order different from the creation order. For example, you are monitoring servers distributed across the globe. Your main datacentre is located in Paris. Something append in New York, and a few milliseconds after something append in Toulouse. Due to location, the event in Toulouse is likely to show up in your datacentre before the New York one. If you analyse data bases on the received time the order will be different than the event time. Computers and network are unreliable. Hence, when temporality is important, you must consider the creation time of the event and not it's received time.

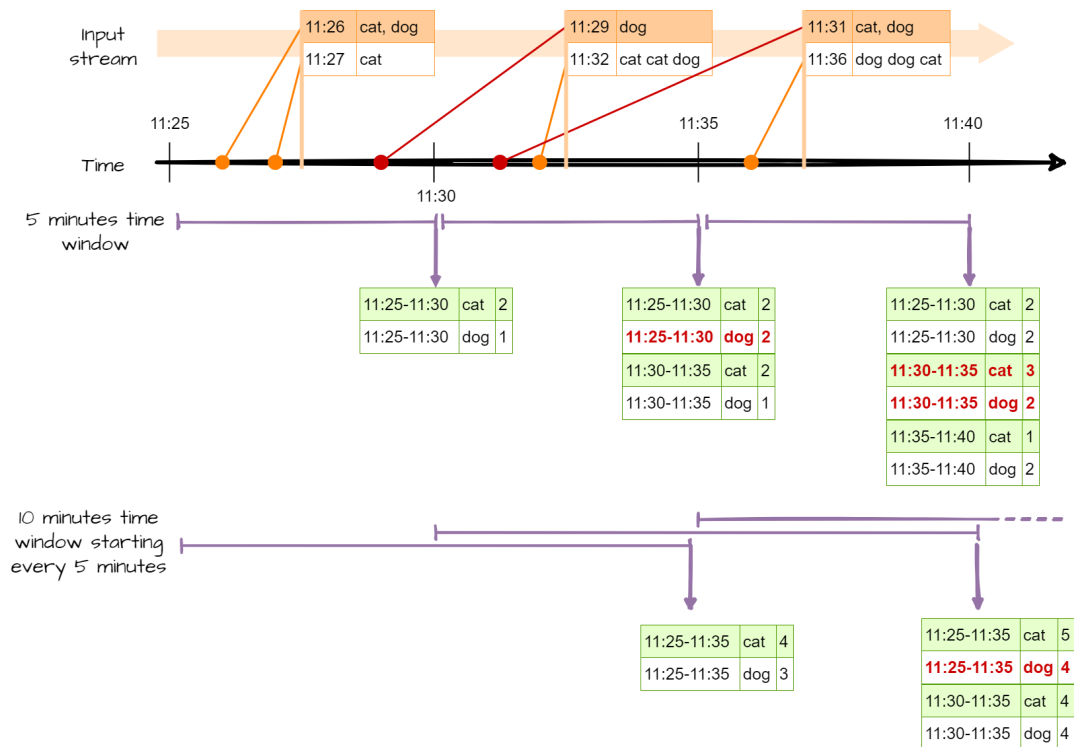
Hopefully, Spark will handle all this complexity for you ! If you have a timestamp column with the event creation spark can update data accordingly to the event time.

For instance is you process some data with a time window, Spark will update the result based on the event-time not the received time. So previous windows can be updated in the future.



To work with time windows, Spark offers two type of windows

- Normal windows. You only consider event in a given windows. All windows are disjoint, and a event is only in one window.
- Sliding windows. You have a fix window size (for example 1 hour) and a timer (for example 10 minute). Every 10 minute, you will process the data with an event time less than 1h.



To create time windows, you need :

- to define a time window : `window(column_with_time_event : str or col, your_time_window : str, timer_for_sliding_window) : str`
- grouping row by event-time using your window : `df.groupBy(window(...))`

To produce the above process :

```

1 # Need some import
2 from pyspark.sql.functions import window, col
3
4 # word count + classic time window
5 df_with_event_time.groupBy(
6     window(df_with_event_time.event_time, "5 minutes"),
7     df_with_event_time.word).count()
8
9 # word count + sliding time window
10 df_with_event_time.groupBy(
11     window(df_with_event_time.event_time, "10 minutes", "5
12     minutes"),
13     df_with_event_time.word).count()

```

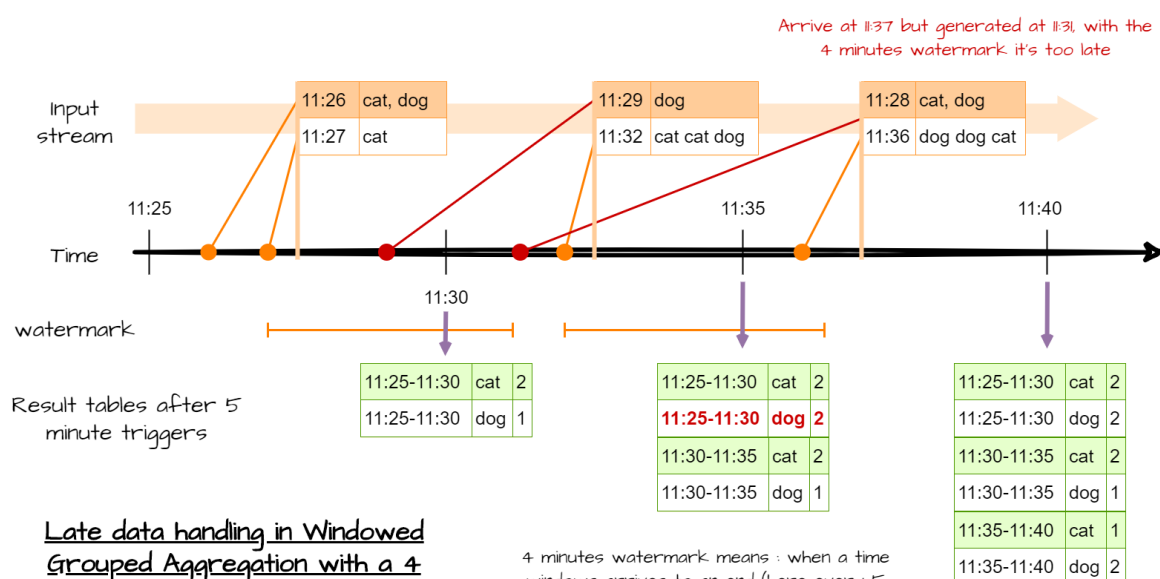
Hand-on 4 : Event-time processing

- Count the number of event with a 1 minute time window (use the `Creation_Time` column)
- Count the number of event by user with a 30 seconds time window (use the `Creation_Time` column)
- Count the number of event by user with a 15 seconds time window sliding every 10 seconds (use the `Creation_Time` column)

Handling late data with watermarks

Processing time event is great, but currently there is one flaw. We never specified how late we expect to see data. This means, Spark will keep some data in memory forever. Because streams never end, Spark will keep in memory every time windows, to be able to update some previous results. But in some cases, you know that after some time, you don't expect new data, or you don't care about it anymore. In other words, after a certain amount of time you want to freeze old results.

Once again, Spark can handle such process, with watermarks.



Late data handling in Windowed Grouped Aggregation with a 4 minutes watermark

4 minutes watermark means : when a time windows arrives to an end (here every 5 minutes), Spark will only process new data that are less late than 4 minutes late compare to the last event time received. In other words, 4 minutes after the last event of a time windows, the result of this window cannot be updated !

To do so, you have to define column as watermark and a the max delay. You have to use the `withwatermark(column, max_delay)` method.

```
1 # Need some import
2 from pyspark.sql.functions import window, col
3
4 # word count + classic time window
5 df_with_event_time.withwatermark(df_with_event_time.event_time, "4
6 minutes")\
7 .groupBy(
8     window(df_with_event_time.event_time, "5 minutes"),
9     df_with_event_time.word).count()
10
11 # word count + sliding time window
12 df_with_event_time.withwatermark(df_with_event_time.event_time, "4
13 minutes")\
14 .groupBy(
15     window(df_with_event_time.event_time, "10 minutes", "5 minutes"),
16     df_with_event_time.word).count()
```

Hand-on 5 : Handling late data with watermarks

- Count the number of event with a 1 minute time window (use the `Creation_Time` column)
- Count the number of event by user with a 30 secondes time window (use the `Creation_Time` column)
- Count the number of event by user with a 15 seconds time window sliding every 10 seconds (use the `Creation_Time` column)

1. À vous de jouer!

Votre but est de construire la première brique dans un tableau de bord pour Wikimedia, la maison mère de Wikipédia, afin de surveiller les articles de l'encyclopédie et de la défendre contre les pillages.

Wikimédia publie un flux de tous les changements qui ont lieu sur l'ensemble des plateformes à l'adresse suivante: <https://stream.wikimedia.org/v2/stream/recentchange>. Le format n'est pas adapté à Spark alors, comme précédemment, **vous devez lancer un serveur qui lit, convertit et transfère le flux vers un port local auquel Spark peut s'abonner** (c'est le serveur `serveur_wikipedia.py`, il communique sur le port 10003!):

Chaque changement est un objet JSON de la forme suivante:

```
1 {
2     "$schema": "/mediawiki/recentchange/1.0.0",
3     "id": 124337776,
4     "type": "categorize",
5     "namespace": 14,
6     "title": "Category:NA-importance India articles",
7     "comment": "[[:Category talk:1990s in Goa]] added to category",
8     "timestamp": 1585047749,
9     "user": "Jevansen",
10    "bot": false,
11    "server_url": "https://en.wikipedia.org",
12    "server_name": "en.wikipedia.org",
13    "server_script_path": "/w",
```

```

14     "wiki": "enwiki",
15     "parsedcomment": "<a href=\"/wiki/Category_talk:1990s_in_Goa\"
title=\"Category talk:1990s in Goa\">Category talk:1990s in Goa</a> added to
category"
16 }

```

Les variables qui nous intéressent sont: `title` (nom de la page), `user` (nom de l'utilisateur), `bot` (est-ce un robot qui a produit le changement), `timestamp` (à quel moment le changement a-t-il été produit), `wiki` (quel site de l'écosystème Wikimedia a été modifié).

1. Stockez ces informations (et uniquement celles-ci) dans un DataSet qui se mettra à jour toutes les 5 secondes
2. Combien de changements sont advenus depuis le début de notre abonnement, sur chaque site de Wikimedia? (vous afficherez le résultat dans la console)
3. Restreignez vous aux données de Wikipédia en français (`wiki=="frwiki"`). Maintenez à jour un DataSet qui donne le nombre d'édition (`type=="edit"`) dans une fenêtre glissante d'une heure calculée toutes les 5 minutes
4. En raison d'un risque élevé de pillage des pages suite à un événement majeur en Haute-Garonne, Wikimedia souhaite mettre en place un suivi des modifications des pages Wikipédia sur ce département. Le fichier `hte-garonne.csv` contient la liste des communes de Haute-Garonne (`communeLabel`), ainsi que le nom de la page Wikipédia correspondante (`articleName`) telle que figurant sur la base de données Wikidata ¹. Combien de modifications ont été effectuées sur l'une de ces pages depuis le début de l'abonnement?

****Vous rendrez votre code au format `.py` sous Moodle.****

```

1  SELECT ?commune ?communeLabel ?articleName WHERE {
2

```

`?commune p:P31 ?estCommune. # P31 = est une instance de`
`?estCommune ps:P31 wd:Q484170. # Q484170 = commune de France`
`?commune wdt:P131 wd:Q12538. # P131 = est situé dans ; Q12538 = Haute-Garonne`

`MINUS{?estCommune pq:P582 ?dateDeFinCommune.} # P582 = a pour date de fin (cas des anciennes communes)`

`?article schema:about ?commune.`
`?article schema:isPartOf https://fr.wikipedia.org/.`
`?article schema:name ?articleName.`
`SERVICE wikibase:label { bd:serviceParam wikibase:language "fr". }`
`}`

```

1
2  ## 4. Expert mode :sparkler:
3

```

```

4 :crossed_swords: En plus des jointure stream x statique, il est également
   possible de faire des jointures entre streams ([pour plus d'info]
   (https://spark.apache.org/docs/latest/structured-streaming-programming-
   guide.html#stream-stream-joins))
5
6 Pensez à lancer le server2 dans serveurs_python/serveur_iot_2 (python
   serveur_iot_2/server2.py)
7
8 ```python
9 # Définition d'un nouveau stream
10 tcp_stream2 = spark\
11     .readStream\
12     .format("socket")\
13     .option("host", "127.0.0.1")\
14     .option("port", "10000")\
15     .load()
16
17 # Schema
18 schema_iot_2 = StructType()\
19     .add('Arrival_Time',LongType(),True)\
20     .add('Creation_Time',LongType(),True)\
21     .add('Device',StringType(),True)\
22     .add('Index',LongType(),True)\
23     .add('Model',StringType(),True)\
24     .add('User',StringType(),True)\
25     .add('bpm',ShortType(),True)\
26
27 # Mise au format plus filtrage
28 filtered_iot_2 = tcp_stream2.selectExpr('CAST(value AS STRING)')\
29     .select(from_json('value', schema_iot_2).alias('json'))\
30     .select('json.*')\
31     .na.drop("any")
32
33 # Jointure des deux flux selon la variable User
34 join_iot = iot_filtered.join(filtered_iot_2, "User").writeStream\
35     .format("console")\
36     .trigger(processingTime='10 seconds')\
37     .start()
38
39 join_iot.stop()

```

☹ Si vous faite bien attention joindre un flux avec un flux et joindre un flux avec des données statiques se font de la même façon car Spark manipule des DataFrame dans les deux cas.

🏠 Compter des mots de citations

🔗 On rentre dans les requêtes vraiment complexes donc ne passez pas beaucoup de temps sur cette partie

Lancer le fichier `server_kaamelott.py` (en vous plaçant dans le dossier `serveur_quote`). Ce serveur envoie des citations de kaamelott aux clients qui s'y connectent (il communique sur l'host 127.0.0.1 et le port 10001). Voici un exemple de donnée qu'il envoie :

```

1 {"character": "Karadoc", "quote": "Quand je pense à la chance que vous avez
   de faire partie d'un clan dirigé par des cerveaux du combat psychologique,
   qui se saignent... aux quatre parfums du matin au soir ! !"}

```


Et voici la requête pour obtenir le nombre de mot reçus par personnage

```
1 #Connexion au stream
2 kaamelott_stream = spark\
3     .readStream.format("socket")\
4     .option("host", "127.0.0.1")\
5     .option("port", "10001").load()
6
7 #Schéma
8 schema_kaamelott = StructType()\
9     .add('character', StringType(), True)\
10    .add('quote', StringType(), True)
11
12 #Application du schéma
13 kaamelott_df = kaamelott_stream.selectExpr('CAST(value AS STRING)')\
14     .select(from_json('value', schema_kaamelott).alias('json'))\
15     .select('json.*')
16
17 # Le comptage de mots
18 # explode/split -> créent plusieurs lignes à partir d'une ligne (ici on
19 # coupe les mots avec split et on fait une ligne par mot de la citation)
20 # groupBy/count : on compte
21 words = kaamelott_df.select(kaamelott_df.character,
22     explode(
23         split(kaamelott_df.quote, ' ')
24     ).alias('word')
25 )\
26     .groupBy(kaamelott_df.character)\
27     .count()\
28     .writeStream\
29     .format("console")\
30     .outputMode("complete")\
31     .trigger(processingTime='5 seconds')\
32     .start()
```

Pour plus d'information :

- [La doc spark officielle](#)
- ZAHARIA, B. C. M. (2018). *Spark: the Definitive Guide.*, O'Reilly Media, Inc. <https://proquest.safaribooksonline.com/9781491912201>
- <https://databricks.com/blog/2018/03/13/introducing-stream-stream-joins-in-apache-spark-2-3.html>
- <https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html>
- <https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html>

Hand-on 1 : Open a stream to a Amazon S3 folder

Open a stream to this folder : `s3://spark-lab-input-data-ensai20202021/Iot/`

- This schema :

```
1  from pyspark.sql.types import StructType, StructField, StringType,
   IntegerType, ArrayType, TimestampType, BooleanType, LongType,
   DoubleType
2
3  schema_tweet = StructType([
4      StructField("auteur", StringType(), True),
5      StructField("contenu", StringType(), True),
6      StructField("date_creation", TimestampType(), True),
7      StructField("entities", StructType([
8          StructField("annotations", ArrayType(StructType([
9              StructField("end", LongType(), True),
10             StructField("normalized_text", StringType(), True),
11             StructField("probability", DoubleType(), True),
12             StructField("start", LongType(), True),
13             StructField("type", StringType(), True)
14         ]), True), True),
15         StructField("cashtags", ArrayType(StructType([
16             StructField("end", LongType(), True),
17             StructField("start", LongType(), True),
18             StructField("tag", StringType(), True)
19         ]), True), True),
20         StructField("hashtags", ArrayType(StructType([
21             StructField("end", LongType(), True),
22             StructField("start", LongType(), True),
23             StructField("tag", StringType(), True)
24         ]), True), True),
25         StructField("mentions", ArrayType(StructType([
26             StructField("end", LongType(), True),
27             StructField("start", LongType(), True),
28             StructField("username", StringType(), True)
29         ]), True), True),
30         StructField("urls", ArrayType(StructType([
31             StructField("description", StringType(), True),
32             StructField("display_url", StringType(), True),
33             StructField("end", LongType(), True),
34             StructField("expanded_url", StringType(), True),
35             StructField("images", ArrayType(StructType([
36                 StructField("height", LongType(), True),
37                 StructField("url", StringType(), True),
38                 StructField("width", LongType(), True)
39             ]), True), True),
40             StructField("start", LongType(), True),
41             StructField("status", LongType(), True),
42             StructField("title", StringType(), True),
43             StructField("unwound_url", StringType(), True),
44             StructField("url", StringType(), True),
45         ]), True), True),
46     ]), True),
47     StructField("hashtags", ArrayType(StringType(), True)),
48     StructField("like_count", LongType(), True),
49     StructField("reply_count", LongType(), True),
```

```

50     StructField("retweet_count", LongType(), True),
51     StructField("others", StructType([
52         StructField("auteur_name", StringType(), True)
53     ]), True),
54 ])

```

- This s3 folder : `s3://spark-lab-input-data-ensai20202021/stream_tweet/`

☹️ Nothing is happening ? That normal Spark still performs lazy evaluation. So if you do not run any action on your data, Spark won't do anything.

3. Printing my first stream

Remember, Spark has two types of methods :

- Transformations which take a DataFrame as input and produce another DataFrame
- And actions, which effectively run computation and produce something, like a file, or an output in your notebook/console.

Processing streams works the same. You still have transformation, the exact same one that can be applied on DataFrame (with some restriction), and actions to output your stream. But because a stream runs continuously, the possible outputs ([sinks](#)) are different. Spark Streaming can :

- Pass data to Kafka
- Write in a file
- Run a foreach function for running arbitrary computation on output records
- Keep data in memory (for testing)
- Print data (for testing)

Moreover, because new data arrive continuously you can

- Append the new transformed data to your output. For instance, you just append new lines to a file
- Only update the relevant line. You compute an aggregate on your data and compute a sum. When a new data arrive, only one line will change. So you just update this new line
- Or you update all the output table every time. Spark can do this only for aggregation queries.

1. La requête pour obtenir ces données depuis <https://query.wikidata.org> est la suivante: [🔗](#)